

BALLOON POPPER!

A step-by-step 2D game development tutorial in Unity



Your goal: build a small, playable game in which the player moves, shoots pins, hits moving balloons, earns points, and can keep playing.

Once the required game works, you can add a title screen, instructions, timer, and game-over flow.

Before You Build: Think Like a Game Designer

- *Balloon Popper!* is a small arcade-style game:
 - The player controls a character along the bottom of the screen and launches pins upward at a balloon that moves automatically.
 - A successful hit gives feedback and increases the score.
 - The challenge comes from timing the shot while the target is moving.

1. Define the game

Design element	Balloon Popper!
Player goal	Hit the moving balloon with a pin and earn points.
Player actions	Move left/right and press Space to shoot.
Game objects	Player, moving Balloon, Pin projectile, background, score display, and optional UI panels.
Rules	The Player moves horizontally within boundaries. The Balloon moves automatically and reverses at the edges. Pins travel upward. A pin that hits the Balloon triggers the pop response and awards one point.
Feedback	The pin disappears on impact, the balloon briefly pops/disappears and returns, the score changes, and an optional sound play.
Game state	The current score and whether the game is playing or over. In the required version, the score is runtime state and resets when the scene/session restarts.
Optional end condition	A countdown reaches zero, or another rule ends the round. This is an extension, not part of the required core loop.

2. The core gameplay loop

A gameplay loop is the repeated sequence of actions and outcomes that keeps a game playable.

In *Balloon Popper!*, the loop is:

Step	What the player/game does	Design purpose
1. Observe	The player watches the balloon move across the screen.	Gives the player a target and a timing problem.
2. Position	The player moves left or right.	Lets the player adjust their location before shooting.
3. Act	The player presses Space to launch a pin.	The player commits to an attempt.
4. Resolve	The pin travels upward. The game checks whether it overlaps the balloon.	Rules and collision detection determine the outcome.
5. Give feedback	On a hit, the balloon pops briefly, the pin is removed, the score increases, and optional sound plays.	Makes success clear and rewarding.
6. Continue	The balloon returns and the player can try again.	Repeats the loop and supports improving timing.

Design check: If the player hits a balloon and the game stops permanently, the loop is broken.

The required version should restore the balloon or create another one so the player can continue playing.

From Game Design to Development

Design describes what should happen. Development turns those rules into GameObjects, components, scripts, UI, and tests.

Game design idea	Unity implementation
Player can move	Player GameObject + Sprite Renderer + PlayerMovement.cs
Balloon moves automatically	Balloon GameObject + Sprite Renderer + BalloonMovement.cs
Player shoots a pin	Pin Prefab + PinMovement.cs + PlayerShoot.cs
Pin hits balloon	Collider2D components + trigger setup + Rigidbody2D + BalloonHit.cs
Hit awards a point	ScoreManager.cs + TextMeshPro score text
Player receives feedback	Balloon pop/respawn, pin removal, optional AudioSource
Game starts/ends (optional)	TitleScene and UI buttons; optional timer/game-over manager

How Unity organizes the game

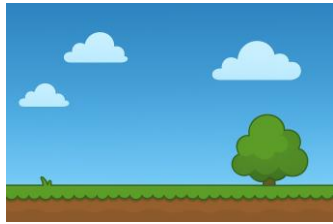
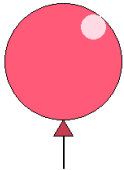
- **Scene:** a **container for the objects used in one part of the game**. The required gameplay scene in *Balloon Popper!* is named GameScene.
- **GameObject:** an **object in the scene**, such as Player, Balloon, Pin, Canvas, or GameManager.
- **Component:** a **feature attached to a GameObject**, such as Transform, Sprite Renderer, Collider2D, Rigidbody2D, or a script.
- **Script:** **C# code that defines behavior**, such as movement, shooting, collision response, or score updates.
- **Prefab:** a **reusable template**. In *Balloon Popper!*, a script: PlayerShoot creates a runtime copy of the Pin Prefab each time Space is pressed.
- **UI: on-screen elements** such as the score, timer, buttons, instructions, and game-over panel.

PART 1 — LAB 1: Build the Foundation

1. Know your assets before you begin

These four image files are supplied for the Balloon Popper project.

Import them into Unity before placing them in the scene.

	<u>Background</u>	The full scene backdrop: sky, clouds, grass, and trees. Place it behind everything else and scale it to fill the camera view.
	<u>Player</u>	The yellow smiley character. It moves left and right when you hold A / Left Arrow or D / Right Arrow.
	<u>Balloon</u>	The pink balloon. It moves left and right automatically and reverses at the boundaries.
	<u>Pin</u>	The projectile artwork. For this lab, turn it into a Prefab asset; shooting it comes in the next lab.

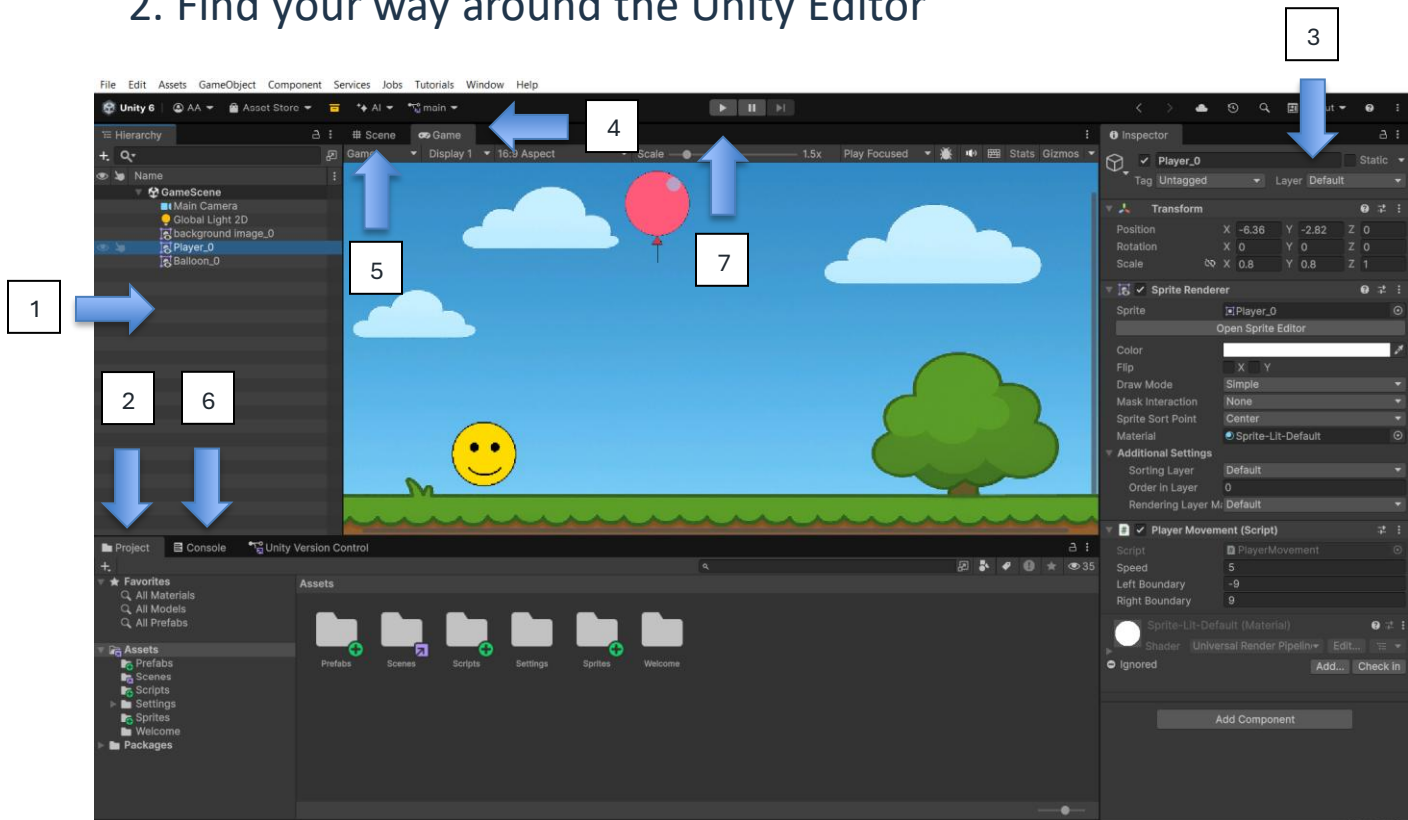
Create this folder structure

```
Assets/  
  Scenes/    → save GameScene here  
  Scripts/   → save C# scripts here  
  Sprites/   → store the imported images here  
  Prefabs/   → store the reusable Pin Prefab here
```

Important: The image file in the Project window is an asset.

When you drag a sprite into the scene, Unity creates a GameObject that uses that asset.

2. Find your way around the Unity Editor



Area	What it does
1. Hierarchy	Lists GameObjects currently in the open scene, such as Main Camera, Background, Player, and Balloon.
2. Project	Stores project files and assets. Open Assets/Sprites or Assets/Scripts here.
3. Inspector	Shows settings and components for the selected object or asset. Attach scripts and edit Speed or boundaries here.
4. Game tab	Shows what the camera displays when the game runs.
5. Scene tab	Use Scene view to position and edit objects in the game world.
6. Console	Shows warnings, debug messages, and errors. Fix red errors before troubleshooting movement.
7. Play button	Click the triangle at the top of Unity to run the game; click again to stop Play Mode.

3. Create the project and prepare Unity

A. Create the project

- Open **Unity Hub** and choose **Projects** → **New Project**.
- Choose a **Universal 2D** template if it is available.
- Name the project **BalloonPopper**, choose a save location, and click **Create Project**.
- Wait for Unity to finish opening and importing.

B. Verify keyboard input before scripting

- Open **Window** → **Package Manager**. Choose Unity Registry and search for **Input System**.
- If it is not installed, select it and click **Install** (or update)
- Open **Edit** → **Project Settings** → **Player**. Find Active Input Handling under Other Settings.
- Choose **Input System Package (New)** or Both. If Unity asks to restart, accept and wait.
- Open **Window** → **General** → **Console** if needed. Check that there are no red compilation errors.

C. Save the scene and make folders

- Choose **File** → **Save As....**. Create **Assets/Scenes** if needed and save the scene as **GameScene**.
- In the Project window, right-click inside **Assets** and create **Scripts**, **Sprites**, and **Prefabs** folders (Scenes may already exist).
- Drag the Background, Player, Balloon, and Pin image files into **Assets/Sprites**.
- Select each image in Project. In the Inspector, set **Texture Type** to **Sprite (2D and UI)**, then click Apply if you change it.

Checkpoint: You can see all four sprites in Assets/Sprites, and GameScene is saved in Assets/Scenes.

4. Build the visible scene

A. Place the background

- In **Assets/Sprites**, drag the **Background** sprite into the **Scene view** or **Hierarchy**.
- Select **Background** and **adjust its Transform Position and Scale until it fills the camera view**.
- In **Sprite Renderer**, set **Order in Layer** to a low value, such as **-10**, **so it appears behind gameplay objects**.

B. Place Player and Balloon

- **Drag the Player sprite into the Scene.**
- **Drag the Balloon sprite into the Scene.**
- Place **Player** near the bottom of the camera view and **Balloon** higher in the play area.
- For **Player** and **Balloon**, set **Sprite Renderer Order in Layer** higher than **Background**, such as **0** or **1**.

C. Check the camera and preview

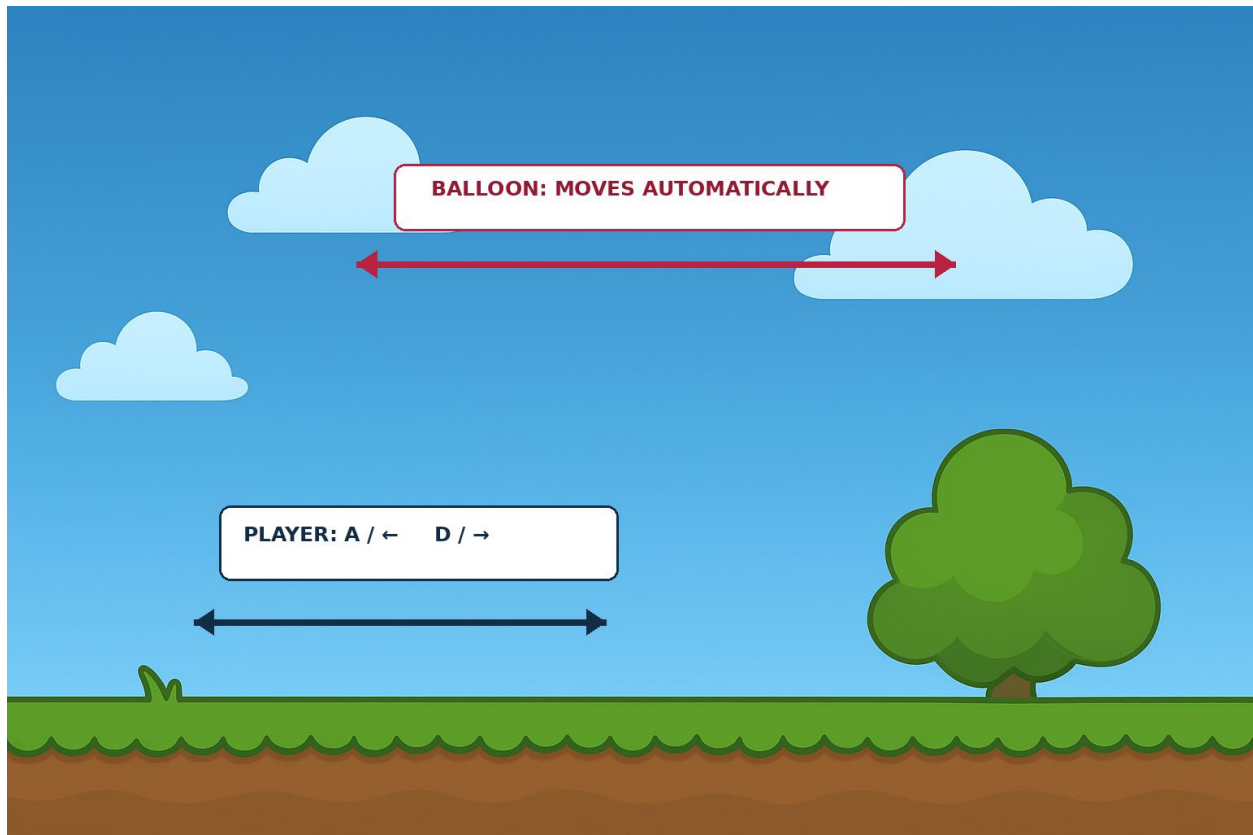
- Select **Main Camera** in the **Hierarchy** and confirm **Projection** is **Orthographic**.
- Adjust **Orthographic Size** if needed so **Background, Player, and Balloon fit in view**.
- Click the **Game** tab. Press **Play** to preview, then press **Play** again to exit **Play Mode** before further edits.

Scene view is for arranging objects.

Game view shows what the camera sees.

If something is missing, check position, scale, sorting order, and camera framing.

5. Understand the two kinds of movement



- **Player → Keyboard-controlled**
 - Hold A or Left Arrow to move left. Hold D or Right Arrow to move right. If no movement key is held, the Player stays still.
- **Balloon → Automatic**
 - The Balloon moves continuously without keyboard input. When it reaches the right boundary, it turns left; when it reaches the left boundary, it turns right.

Boundaries: Both scripts use X-axis limits. These numbers depend on your camera framing and sprite size. The sample code starts with -7 and +7; adjust them if your scene uses different dimensions.

Before testing keyboard-controlled movement, click inside the Game view so Unity sends key input to the game.

6. Create and attach PlayerMovement.cs

Create the script

- In the Project window, open **Assets/Scripts**.
- Right-click an empty area and choose **Create → Scripting → MonoBehaviour Script** (or Create → C# Script (menu wording varies by Unity version)).
- Name it exactly **PlayerMovement**. Open it in your code editor.
- Replace the template with the code below, save the file, and return to Unity. Wait for compilation to finish.

```
using UnityEngine;
using UnityEngine.InputSystem; // Lets us read keyboard input.

public class PlayerMovement : MonoBehaviour {

    // Editable in the Inspector.
    [SerializeField] private float speed = 5f;
    [SerializeField] private float leftBoundary = -7f;
    [SerializeField] private float rightBoundary = 7f;

    void Update() {

        // Stop safely if no keyboard device is available.
        if (Keyboard.current == null) return;

        // 0 = still, -1 = left, +1 = right.
        float direction = 0f;

        // A or Left Arrow moves left.
        if (Keyboard.current.aKey.isPressed ||
            Keyboard.current.leftArrowKey.isPressed) {
            direction = -1f;
        }

        // D or Right Arrow moves right.
        else if (Keyboard.current.dKey.isPressed ||
            Keyboard.current.rightArrowKey.isPressed) {
            direction = 1f;
        }
    }
}
```

```

        // Copy the current position.
        Vector3 position = transform.position;

        // Move horizontally; deltaTime keeps speed frame-rate
        independent.
        position.x += direction * speed * Time.deltaTime;

        // Keep the Player inside the horizontal limits.
        position.x = Mathf.Clamp(position.x, leftBoundary,
        rightBoundary);

        // Apply the new position.
        transform.position = position;
    }
}

```

What the important lines do

Code / idea	Meaning
Keyboard.current...isPressed	Checks whether a key is currently held down.
direction = -1f / 1f	Stores which horizontal direction to move. Zero means no movement.
speed * Time.deltaTime	Controls movement speed while keeping it consistent across different frame rates.
Mathf.Clamp(...)	Limits the Player's X position between leftBoundary and rightBoundary.
transform.position = position	Applies the calculated position to the GameObject.

7. Attach and test PlayerMovement

- Return to Unity and select **Player** in the Hierarchy.
- Click **Add Component** in the Inspector, search for **PlayerMovement**, and add it. Alternatively, you can **drag the script from Assets/Scripts onto Player**.
- Set Speed to 5. Set Left Boundary and Right Boundary to values that keep Player visible. Start with -7 and 7 only if they match your camera framing.
- Save the scene with Ctrl+S (Windows) or Cmd+S (Mac).
- Click Play, then click inside the Game view. Hold A / Left Arrow and D / Right Arrow to test both directions.
- Try to move beyond both edges. Player should stop at the boundaries. Click Play again to exit Play Mode.

Player test checklist

- The Player moves left with A and Left Arrow.
- The Player moves right with D and Right Arrow.
- The Player stops when no movement key is pressed.
- The Player cannot move past the configured horizontal boundaries.

If it does not work: Check the Console for red errors, confirm the filename and class are both PlayerMovement, confirm the script is attached to Player, verify Input System settings, and click inside Game view before testing.

8. Create and attach BalloonMovement.cs

Create the script

- In **Assets/Scripts**, create a new C# or MonoBehaviour script named **BalloonMovement.cs**:

```
using UnityEngine;

public class BalloonMovement : MonoBehaviour {

    // Editable in the Inspector.
    [SerializeField] private float speed = 3f;
    [SerializeField] private float leftBoundary = -7f;
    [SerializeField] private float rightBoundary = 7f;

    // True = moving right; false = moving left.
    private bool movingRight = true;

    void Update() {

        // Convert the stored direction into +1 or -1.
        float direction = movingRight ? 1f : -1f;

        // Copy the current position and move along X.
        Vector3 position = transform.position;
        position.x += direction * speed * Time.deltaTime;

        // At the right edge, clamp and turn left.
        if (position.x >= rightBoundary) {
            position.x = rightBoundary;
            movingRight = false;
        }

        // At the left edge, clamp and turn right.
        else if (position.x <= leftBoundary) {
            position.x = leftBoundary;
            movingRight = true;
        }

        // Apply the new position.
        transform.position = position;
    }
}
```

How the automatic movement works

Code / idea	Meaning
<code>movingRight = true</code>	The Balloon starts by moving right.
<code>movingRight ? 1f : -1f</code>	If <code>movingRight</code> is true, direction is +1; otherwise direction is -1. This is the ternary operator, a short if/else expression.
<code>position.x += ...</code>	Updates the Balloon's horizontal position every frame.
<code>position.x >= rightBoundary</code>	At the right limit, place the Balloon at the limit and set <code>movingRight</code> to false.
<code>position.x <= leftBoundary</code>	At the left limit, place it at the limit and set <code>movingRight</code> to true.

9. Attach, test, and adjust BalloonMovement

- Select **Balloon** in the Hierarchy.
- Click **Add Component**, search for **BalloonMovement**, and add it.
- Set Speed to 3 to start. Adjust Left Boundary and Right Boundary so the entire Balloon stays visible. The sample -7 and +7 may need adjustment.
- Save the scene. Click Play and watch the Balloon move without any keyboard input.
- Confirm that it reverses at both boundaries and that Player still moves independently.
- Exit Play Mode before editing the scene.

Making a speed change stick

If you change Speed while Play Mode is active, Unity usually reverts that Inspector change when you stop.

To keep a tested value, stop Play Mode and enter the value again. To change the script default instead, edit the speed number in `BalloonMovement.cs`, save, and return to Unity.

Unity may keep the old Inspector value; re-enter your chosen value or use the component menu → Reset. Reset restores all component defaults, so note your boundary values first.

If the Balloon does not reverse: Check that BalloonMovement is attached to Balloon, the Console has no red errors, and Left Boundary is less than Right Boundary. Adjust limits to account for the Balloon's width.

10. Create the Pin Prefab

- In the Project window, open **Assets/Sprites**.
- **Drag the Pin sprite into the Hierarchy or Scene.** Rename the new GameObject **Pin**. Adjust its Transform Scale so it is small compared with the Balloon. If needed, rotate it so the tip points upward.
- **Drag Pin from the Hierarchy into Assets/Prefabs.** Unity creates a Prefab asset. **Confirm a Pin asset appears in Assets/Prefabs.**
- **Delete the temporary Pin GameObject from the Hierarchy.** Keep the Prefab asset in the Project window.

A Prefab is a reusable template.

The temporary Pin in the Hierarchy is a scene instance; the Pin asset in Assets/Prefabs is the saved template. Shooting the Pin is part of a later lab.

11. Final checklist and troubleshooting

Check	Expected result
Project folders	Assets contain Scenes, Scripts, Sprites, and Prefabs.
GameScene	Saved at Assets/Scenes/GameScene.unity.
Sprites	Background, Player, Balloon, and Pin are imported in Assets/Sprites.
Visible scene	Background, Player, and Balloon are visible in Game view.
PlayerMovement	A/D and arrow keys move Player horizontally; boundaries keep it in view.
BalloonMovement	Balloon moves on its own and reverse at both edges.
Pin Prefab	Pin asset is in Assets/Prefabs; temporary Pin instance is removed from scene.
Saved work	Scene is saved and Console has no red errors.

Before you finish: Save the scene and project.

Remember where the project folder is stored, and reopen this same project for the next lab.

Lab 1's stopping point is a working Player, an automatically moving Balloon, and a saved Pin Prefab—not shooting, collision detection, or scoring.

PART 2 — LAB 2: Shooting, Collision, and Score

Continue in the same BalloonPopper project from Lab 1. Do not create a new project. Keep PlayerMovement.cs and BalloonMovement.cs; Lab 2 adds four required scripts.

Lab 2 required outcome

- Move horizontally, shoot pins with Space, hit the moving balloon, see the balloon pop, gain one point, and continue playing.

Script	Attach to / connect with	Responsibility
PlayerMovement.cs (Lab 1)	Player	Moves the Player horizontally; keep it.
BalloonMovement.cs (Lab 1)	Balloon	Moves the Balloon automatically; keep it.
PinMovement.cs (Lab 2)	Pin Prefab	Moves each pin upward and removes it off-screen.
PlayerShoot.cs (Lab 2)	Player	Reads Space and creates a copy of the Pin Prefab.
BalloonHit.cs (Lab 2)	Balloon	Detects a Pin trigger hit, gives pop/respawn feedback, and calls score/audio references.
ScoreManager.cs (Lab 2)	GameManager object + ScoreText	Tracks the in-session score and updates the visible UI.

Step 1. Recheck the Lab 1 project

- Open the existing BalloonPopper project and GameScene. Confirm the background and Player are visible in Game view.
- Test the Player's left/right controls and the Balloon's automatic movement.
- Confirm the Pin Prefab is in Assets/Prefabs and the two Lab 1 scripts work without Console errors. Stop Play Mode before making changes.

Step 2. Prepare the Pin Prefab

- In Project, open **Assets/Prefabs** and select the **Pin Prefab**. Confirm its Sprite Renderer uses the Pin sprite.
- Add a **Box Collider 2D** that reasonably fits the pin. Enable **Is Trigger**.
- Add **Rigidbody 2D** if neither the Pin nor Balloon already has one. For the **Pin**, set **Gravity Scale** to **0**.
- Create a **tag** named **Pin** if it does not already exist, then set the **Pin Prefab's Tag** to **Pin**. Tag capitalization must match.
- Attach **PinMovement.cs** using the code below. Set Speed to about 8 and Destroy Y to a value above the camera view (often around 6; adjust to your scene).

```
using UnityEngine;
public class PinMovement : MonoBehaviour {

    // Editable in the Inspector: how fast the pin travels upward.
    [SerializeField] private float speed = 8f;
    // Remove the pin when it goes above this Y coordinate.
    [SerializeField] private float destroyY = 6f;

    void Update() {

        // Move upward each frame. deltaTime helps keep movement
        // consistent across computers with different frame rates.
        transform.position += Vector3.up * speed * Time.deltaTime;
        // Check whether the pin has left the visible play area.
        if (transform.position.y > destroyY) {
            // Destroy only this spawned pin, not the Prefab asset.
            Destroy(gameObject);
        }
    }
}
```

Test checkpoint: A temporary Pin object placed in the scene should travel upward and disappear after passing Destroy Y.

Delete any temporary scene Pin after testing; shooting will create pins later.

Step 3. Add shooting to Player

- Create **PlayerShoot.cs** in **Assets/Scripts** and attach it to **Player**.
- In the Inspector, drag the **Pin Prefab** asset from Project into **the Pin Prefab field**. Do not assign a temporary Pin object from Hierarchy.
- Leave Launch Offset at X = 0, Y = 0.6, Z = 0 initially. Increase Y if the pin spawns inside the Player sprite.
- Press Play, click inside Game view, move the Player, and press Space. Each press should create a pin that travels upward.

```
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerShoot : MonoBehaviour {

    // Reusable Pin Prefab assigned through the Inspector.
    [SerializeField] private GameObject pinPrefab;
    // Spawn the pin slightly above the player's position.
    [SerializeField] private Vector3 launchOffset = new Vector3(0f,
0.6f, 0f);

    void Update(){

        // Keyboard.current is null if no keyboard is available.
        if (Keyboard.current == null)
            return;

        // True only on the frame Space is first pressed.
        if (Keyboard.current.spaceKey.wasPressedThisFrame) {

            // Avoid trying to create an object if the field is empty.
            if (pinPrefab != null) {
                // Create a new copy at the player position plus
offset.
                Instantiate(
                    pinPrefab,
                    transform.position + launchOffset,
                    Quaternion.identity
                );
            }
        }
    }
}
```

```
        else {  
            // Make a missing Inspector assignment easy to  
diagnose.  
            Debug.LogWarning("Assign the Pin Prefab in the  
Inspector.");  
        }  
    }  
}
```

If Space does nothing: click Game view, confirm PlayerShoot is attached to Player, assign the Pin Prefab, check Input System settings, and fix any red Console errors.

Step 4. Configure collision detection

- Stop Play Mode. Select **Balloon** and add a **CircleCollider2D** if it does not have one. Fit the collider to the balloon sprite and enable **Is Trigger**.
- Confirm the **Pin Prefab** has a **BoxCollider2D** with **Is Trigger enabled**.
 - At least one of the two objects must have a Rigidbody2D; for this simple game, **Rigidbody2D on the Pin with Gravity Scale = 0** is a convenient option.
- Confirm the **Pin Prefab tag** is exactly **Pin**.
- Create **BalloonHit.cs**, attach it to **Balloon**, and use the code below. Leave Respawn Delay around 0.25 seconds for now.

```
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;

public class BalloonHit : MonoBehaviour {

    // Drag the ScoreManager component into this field later.
    [SerializeField] private ScoreManager scoreManager;

    // AudioSource that plays the pop sound.
    [SerializeField] private AudioSource popAudio;

    // How long the balloon stays hidden after a hit.
    [SerializeField] private float respawnDelay = 0.25f;

    // Stops several pins from scoring during one pop.
    private bool alreadyPopped;

    // Components that are hidden/disabled during the pop.
    private Collider2D balloonCollider;
    private SpriteRenderer balloonRenderer;

    void Awake(){

        // Cache components attached to this same Balloon object.
        balloonCollider = GetComponent<Collider2D>();
        balloonRenderer = GetComponent<SpriteRenderer>();

    }
```

```

void OnTriggerEnter2D(Collider2D other) {

    // Ignore collisions that are not with a Pin.
    if (!other.CompareTag("Pin") || alreadyPopped)
        return;

    alreadyPopped = true;

    // Award a point if the ScoreManager has been assigned.
    if (scoreManager != null)
        scoreManager.AddPoint();

    // Play sound if the AudioSource has been assigned.
    if (popAudio != null)
        popAudio.Play();

    // Remove the pin that made contact with the balloon.
    Destroy(other.gameObject);

    // Hide the balloon, wait briefly, then restore it.
    StartCoroutine(PopAndRespawn());

}

IEnumerator PopAndRespawn() {

    // Turn off the collider to prevent duplicate hits while
hidden.
    if (balloonCollider != null)
        balloonCollider.enabled = false;

    // Turn off the sprite so the pop is visible to the player.
    if (balloonRenderer != null)
        balloonRenderer.enabled = false;

    // Wait for the configured respawn delay.
    yield return new WaitForSeconds(respawnDelay);
    // Restore the sprite and collision for the next shot.
    if (balloonRenderer != null)
        balloonRenderer.enabled = true;
    if (balloonCollider != null)
        balloonCollider.enabled = true;
    alreadyPopped = false;

}
}

```

Test checkpoint: Shoot the Balloon.

The Pin should disappear and the Balloon should briefly disappear, then return.

If nothing happens, check both Collider2D components, Is Trigger, Rigidbody2D, the Pin tag, and the Console.

Step 5. Add the score display

- Stop Play Mode. In Hierarchy, right-click → **UI** → **Canvas**. If prompted, **import TMP Essentials**.
- Right-click **Canvas** → **UI** → **Text - TextMeshPro**. Rename it **ScoreText**. Set its text to **Score: 0**, choose a readable font size (for example, 32–40), and position it near the top of the screen.
- Create an **empty GameObject** in Hierarchy named **GameManager**.
- Create **ScoreManager.cs** and attach it to **GameManager**.
- Drag **ScoreText** from Hierarchy into **ScoreManager's Score Text** field.
- Select **Balloon** and drag **GameManager** into **BalloonHit's Score Manager** field.
- Press Play and hit the Balloon. The display should change from Score: 0 to Score: 1 for each successful hit.

```
using TMPro;
using UnityEngine;
public class ScoreManager : MonoBehaviour {

    // UI label assigned in the Inspector.
    [SerializeField] private TMP_Text scoreText;

    // Score for the current play session.
    private int score = 0;

    // Read-only score access for the optional game-over screen.
    public int CurrentScore => score;

    void Start(){
        // Display the initial score when the scene begins.
        UpdateScoreDisplay();
    }
}
```

```

public void AddPoint(){
    // Add one for each successful balloon hit.
    score++;

    // Refresh the visible score.
    UpdateScoreDisplay();
}

public void ResetScore(){
    // Reset the score when starting a fresh round.
    score = 0;
    UpdateScoreDisplay();
}

void UpdateScoreDisplay(){
    // Avoid a null-reference error if no text was assigned.
    if (scoreText != null)
        scoreText.text = "Score: " + score;
    }
}

```

Score is runtime game state: it exists during the current session and resets when the scene is reloaded.

Saving a high score between sessions is a separate persistence feature.

Step 6. Adding audio on pop

- Download the [Balloon Pop audio file](#), select **Balloon** and add an **Audio Source**.
- Assign the sound to **Audio Generator**. Turn off **Play On Awake**. Leave Spatial Blend at 0 (2D) if the sound should be equally audible throughout the scene.
- Drag **Balloon's AudioSource component** into **BalloonHit's Pop Audio field**.
- Press Play and hit the Balloon to test. If no sound asset is supplied, skip this step; scoring and hit response must still work.

Step 7. Test the complete required loop

- Player moves left and right within the intended area.
- Space creates a Pin from the Prefab at or slightly above Player.
- Each Pin travels upward and disappears off-screen.
- A Pin hitting the Balloon triggers the pop response; the Player is not destroyed.
- The hit Pin disappears; the score increases exactly once.
- The Balloon returns and the player can shoot again.
- Optional sound plays only when the Balloon is hit.
- The Console has no red errors.

Once this is complete, **the following features are optional**.

Add them only after the required Lab 1 + Lab 2 game works.

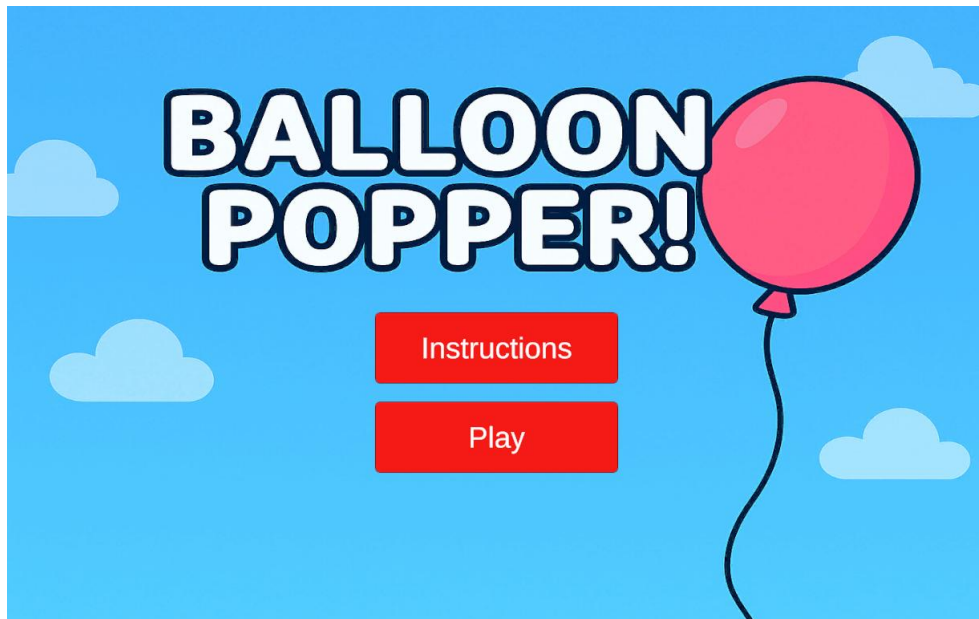
You can choose one extension or combine several if you have time.

PART 3 — OPTIONAL EXTENSIONS

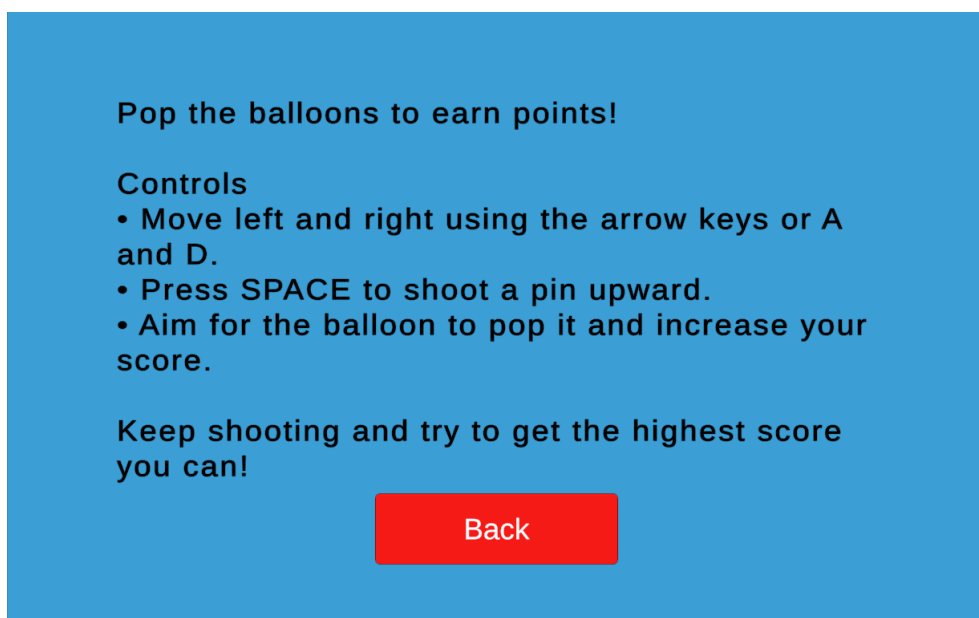
Optional A. Title screen and instructions

This optional extension adds a **Title screen** and a **Instructions panel**.

Example title screen: uses the supplied title-screen image as the scene background. Add the Instructions and Play buttons on top of it;



Example Instructions panel: explain the objective and controls, with a Back button to return to the title screen.



Step 1. Create the Title Scene

- In the **Project** window, open **Assets/Scenes**.
- Right-click inside the folder and select **Create → Scene**.
- Name the new scene **TitleScene**.
- Double-click **TitleScene** to open it.
- Save the scene using **File → Save**.

Step 2. Add the Title-Screen Background

- Download the supplied [title-screen image](#).
- Import the image into your Unity project by dragging it into the Assets folder.
- Select the image in the **Project** window.
- In the **Inspector**, set **Texture Type** to **Sprite (2D and UI)**, then click **Apply**.
- Drag the image from the Project window into the **Scene** view.
- Select the image in the **Hierarchy** and adjust its position and scale so that it fills the camera's view.
- Open the **Game** tab to check how the background looks from the camera. Adjust it as needed.
- **Important:** Use the supplied image directly as the background. Do not add another title image or a separate title-text panel on top of it.

Step 3. Create the Canvas and Buttons

- In the **Hierarchy**, right-click an empty area.
- Select **UI → Canvas**. Unity will also create an EventSystem if one is needed.
- Select the Canvas and check that its **Canvas Scaler** is configured to scale the UI with the screen size:
 - **UI Scale Mode:** Scale With Screen Size
 - **Reference Resolution:** For example, 800 × 600 for a landscape game
 - **Screen Match Mode:** Match Width Or Height
 - **Match:** 0.5

- Right-click the Canvas and select **UI → Button - TextMeshPro**.
- Rename the button **PlayButton**.
- Select its child text object and change the displayed label to **Play**.
- Create a second **Button - TextMeshPro** under the Canvas.
- Rename it **InstructionsButton** and change its displayed label to **Instructions**.
- Use each button's **Rect Transform** to position and size it appropriately over the background image.
- Open the **Game** tab to check that both buttons are visible and readable.

Step 4. Create the Instructions Panel

- In the **Hierarchy**, right-click the Canvas.
- Select **UI → Panel** and rename it InstructionsPanel.
- Resize the panel to cover the area where you want the instructions to appear. You may use a partially transparent background so the title image remains visible behind it.
- Right-click InstructionsPanel and select **UI → Text - TextMeshPro**.
- Edit the text to display the following instructions:

How to Play:

Move with A / Left Arrow and D / Right Arrow.

Press Space to shoot pins. Hit balloons to earn points.
- Adjust the text object's Rect Transform so the instructions fit inside the panel.
- Right-click InstructionsPanel and select **UI → Button - TextMeshPro**.
- Rename the button BackButton and change its displayed label to **Back**.
- Position the Back button beneath the instructions.
- Select InstructionsPanel in the Hierarchy and **uncheck the checkbox next to its name in the Inspector**. This disables the panel so it starts hidden when the title scene opens.

Step 5: Create the Title Menu Controller Script

- The controller script connects the buttons to their actions: starting the game, opening the instructions, and returning to the title screen.
- In the **Project** window, open your Assets/Scripts folder.
- Right-click inside the folder and select **Create → Scripting → MonoBehaviour Script** (the menu wording may differ between Unity versions).
- Name the script **TitleMenuController**.
- Double-click the script to open it in your code editor.
- Replace the existing code with the following:

```
using UnityEngine;
using UnityEngine.SceneManagement;
public class TitleMenuController : MonoBehaviour {

    // Assign the instructions panel by dragging it from the
Hierarchy.
    [SerializeField] private GameObject instructionsPanel;

    // This must exactly match the name of your gameplay scene.
    [SerializeField] private string gameSceneName = "GameScene";

    void Start() {

        // Make sure the instructions are hidden when the title scene
opens.
        if (instructionsPanel != null)
            instructionsPanel.SetActive(false);

    }

    public void StartGame(){

        // Load the gameplay scene when START is clicked.
        SceneManager.LoadScene(gameSceneName);

    }
}
```

```

public void ShowInstructions() {

    // Show the instructions panel.
    if (instructionsPanel != null)
        instructionsPanel.SetActive(true);

}

public void ShowTitle() {

    // Hide instructions to reveal the title image and its buttons
again.
    if (instructionsPanel != null)
        instructionsPanel.SetActive(false);

}
}

```

Step 6. Attach the Script and Connect the Buttons

A. Attach the script to a GameObject

- In the Hierarchy, right-click an empty area and select **Create Empty**.
- Rename the new GameObject **TitleMenuController**.
- With the GameObject selected, click **Add Component** in the Inspector.
- Search for **TitleMenuController** and **attach the script**.
- In the Inspector, find the **Instructions Panel field** on the script component.
- **Drag InstructionsPanel from the Hierarchy into this field.**

B. Connect the Play button

- Select **PlayButton** in the Hierarchy.
- In the Inspector, find the **Button component** and expand **On Click()**.
- **Click + to add an event.**
- Drag the **TitleMenuController** GameObject from the Hierarchy into the empty object field.

- Open the function dropdown and select **TitleMenuController → StartGame()**.

C. Connect the Instructions button

- Select **InstructionsButton** in the Hierarchy.
- Under the **Button component's On Click()** section, click +.
- Drag the **TitleMenuController** GameObject into the empty object field.
- Select **TitleMenuController → ShowInstructions()** from the function dropdown.

D. Connect the Back button

- If **InstructionsPanel** is disabled, **enable it temporarily in the Hierarchy so you can select BackButton**.
- Select **BackButton** and find its **Button component's On Click()** section.
- **Click +** and **drag the TitleMenuController** GameObject into the empty object field.
- Select **TitleMenuController → ShowTitle()** from the function dropdown.
- If you enabled **InstructionsPanel** to configure the button, **disable** it again in the Hierarchy.

E. Test the buttons

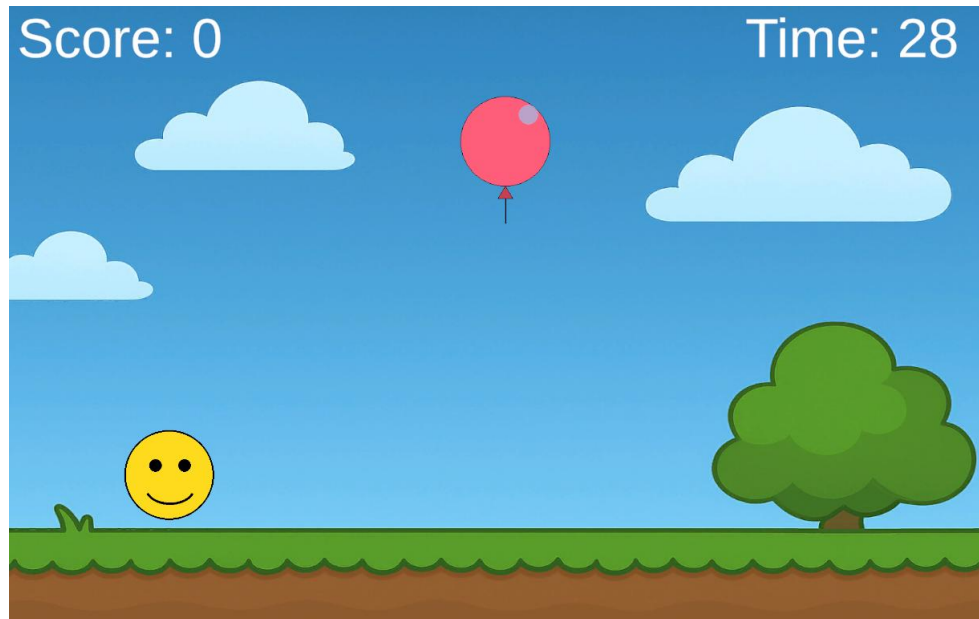
- Save the scene.
- Press Play in Unity.
- Click Instructions. The instructions panel should appear.
- Click Back. The panel should disappear, revealing the title screen.
- Click Play. The game should load GameScene.

If a button does not work, check that its On Click() event references the TitleMenuController GameObject and the correct method.

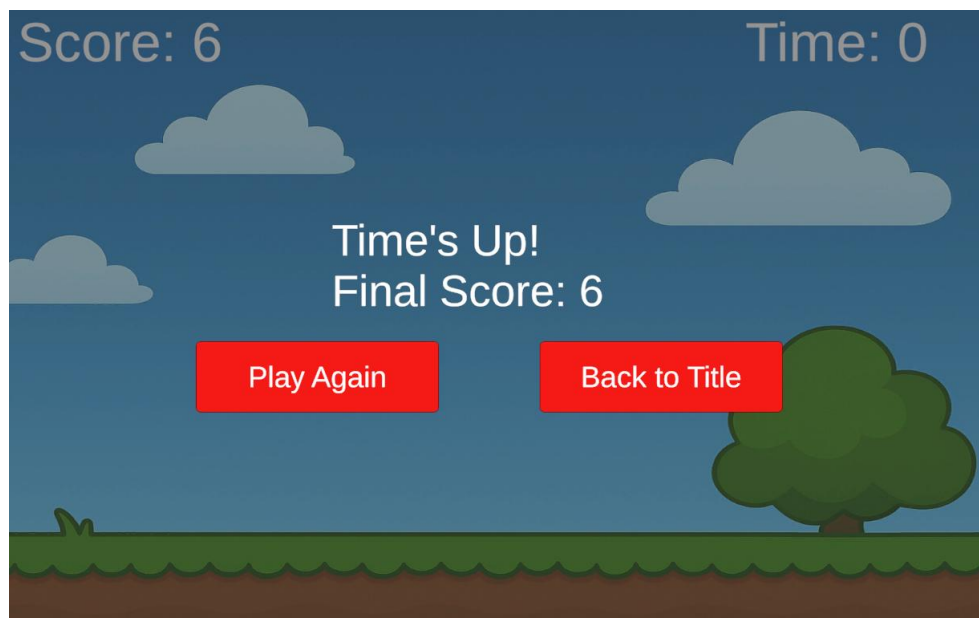
Optional B. Timer and game-over screen

This optional extension adds a **countdown timer** and a **Game Over screen**.

Example gameplay screen: Score and Timer appear at the top while the Player moves along the ground and shoots pins upward.



Example screen: When time runs out, the game stops, the final score is displayed, and the player can choose to play again or return to the title screen.



Step 1. Organize the gameplay objects

- Open **GameScene** in Unity.
- In the Hierarchy, create an **empty GameObject** named **GameplayRoot**.
- **Drag Player and Balloon under GameplayRoot** so they become its children.
- Keep **GameManager** and **Canvas** outside **GameplayRoot**. These objects need to remain active after gameplay ends so the timer and Game Over UI can function.

Step 2. Create the timer and Game Over UI

Create the timer:

- Select **Canvas** in the Hierarchy.
- Right-click Canvas and choose **UI → Text - TextMeshPro**.
- Name the new text object **TimerText**.
- Set its displayed text to **Time: 30**.
- Position it near **ScoreText**, such as in the upper-right corner of the screen.

Create the Game Over screen:

- Right-click Canvas and choose **UI → Panel**. Name it **GameOverPanel**.
- Inside the panel, create a **TextMeshPro text object** named **FinalScoreText**.
- Add two buttons labeled **Play Again** and **Back to Title**.
- Select **GameOverPanel** in the Hierarchy and uncheck the checkbox next to its name at the top of the Inspector. The panel should be **inactive** when the game starts.

Step 3. Create and attach GameManager.cs

- Create **GameManager.cs** in **Assets/Scripts** or open the existing file. Replace its contents with the complete script below.

- Attach **GameManager.cs** to the existing **GameManager GameObject**—the same object that already has ScoreManager. Do not create a second GameManager.

```
using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour {

    [Header("Scene Names")]

    [SerializeField] private string titleSceneName = "TitleScene";
    [SerializeField] private string gameSceneName = "GameScene";

    [Header("Gameplay and UI")]

    [SerializeField] private GameObject gameplayRoot;
    [SerializeField] private GameObject gameOverPanel;
    [SerializeField] private TMP_Text timerText;
    [SerializeField] private TMP_Text finalScoreText;
    [SerializeField] private ScoreManager scoreManager;

    [Header("Round Settings")]

    [SerializeField] private float roundDuration = 30f;
    private float timeRemaining;
    private bool isGameOver;

    void Start() {

        // Ensure the game is not frozen from a previous scene.
        Time.timeScale = 1f;
        timeRemaining = roundDuration;
        isGameOver = false;

        // The panel should be hidden when the round begins.
        if (gameOverPanel != null)
            gameOverPanel.SetActive(false);

        UpdateTimerText();
    }
}
```

```

void Update() {

    // Do not count down after the round has ended.
    if (isGameOver)
        return;

    timeRemaining -= Time.deltaTime;

    if (timeRemaining <= 0f) {

        timeRemaining = 0f;
        EndGame();

    }

    UpdateTimerText();
}

void UpdateTimerText() {

    if (timerText != null)
        timerText.text = "Time: " +
Mathf.CeilToInt(timeRemaining);

}

void EndGame() {

    isGameOver = true;

    // Stop time-based gameplay movement.
    Time.timeScale = 0f;

    if (finalScoreText != null && scoreManager != null) {

        finalScoreText.text =
            "Time's Up! Final Score: " + scoreManager.CurrentScore;

    }

    if (gameOverPanel != null)
        gameOverPanel.SetActive(true);
}

```

```

        // Optional: hide the gameplay objects behind the panel.
        if (gameplayRoot != null)
            gameplayRoot.SetActive(false);
    }

    public void PlayAgain() {

        // Restore time before reloading the gameplay scene.
        Time.timeScale = 1f;
        SceneManager.LoadScene(gameSceneName);

    }

    public void BackToTitle() {

        // Always restore time before changing scenes.
        Time.timeScale = 1f;
        SceneManager.LoadScene(titleSceneName);

    }

    void OnDisable() {

        // Prevent the game remaining frozen after scene changes.
        Time.timeScale = 1f;

    }
}

```

Important: This timer/game-over extension is separate from the required score loop. The current score is not saved between sessions. If you add a title scene, ensure scene names match exactly and both scenes are included in the build.

PART 4 — BUILD FOR THE WEB AND PUBLISH ON ITCH.IO

Once your game works in Unity, build it for the Web so other people can play it in a browser.

Step 1. Save and add your scenes to the build

- Before building, **make sure your scenes are saved**. Unity needs to know which scenes to include in the published game and which scene should open first.
- Open **File** → **Build Profiles** in Unity 6, or **File** → **Build Settings** in some older Unity versions.
- Find the **Scene List** or **Scenes in Build** section.
- Add your scenes:
 - If you have a title screen: **Add both TitleScene and GameScene. Put TitleScene first** in the list so the game opens at the title screen.
 - If you do not have a title screen: **Add GameScene only**.
- If a scene is missing from the list, open that scene and use **Add Open Scenes**, or **drag the scene asset from Assets/Scenes into the list**. The exact control may vary by Unity version.
- Confirm that every scene your game loads is included. For example, if your title screen loads GameScene, that scene must also be in the build.
- **Important: The scene order determines which scene opens first when the game launches.** The scene names used in your scripts must also match the names of the saved scenes.

Step 2. Select the Web build configuration

- In Build Profiles or Build Settings, select **Web** as the target platform.
- If your Unity version provides the Web configuration choices, select **Web → Desktop → Release**. Web is the platform; Desktop and Release are configuration options. Do not select the Windows, Mac, or Linux desktop platform.

- If Web Build Support is missing, install it for your Unity Editor version through Unity Hub.
- For a release build, make sure **Development Build** is turned off, if that option is available.
- Save your project.
- The exact menu labels may differ between Unity versions. Keep the target platform set to Web.

Step 3. Build your game

- Click **Build** (or **Build and Run** if you want Unity to build and test the game).
- Choose a new, empty output folder, such as **BalloonPopperWebBuild**. Do not use your Unity project folder as the output folder.
- Wait for Unity to finish building.
- Open the output folder and confirm it contains **index.html** and the supporting build files and folders, such as **Build** and **TemplateData**.

Step 4. Create a ZIP File

- itch.io needs the complete Web build, not just the HTML file or your Unity project.
 - Open the folder Unity created for the build.
 - Select **the contents of the build folder**, including **index.html**, **Build**, **TemplateData**, and any other files Unity generated.
 - Compress the selected contents into a ZIP file.
 - Open the ZIP to check its structure. **index.html** should be at the top level, not buried inside an extra folder.
- Keep your editable Unity project as a separate backup so you can continue working on your game.

Step 5. Upload and submit to the CISC 3667 Game Jam

- Sign in to itch.io and open the [CISC 3667 F26 Unity Labs itch.io Game Jam](#) page.
- Follow the jam's instructions to join the jam, if you have not already done so.

- Create a project page for your game or open your existing project page.
- Set the **project type to HTML**.
- **Upload the ZIP file containing your Web build.**
- **Enable the option indicating that the uploaded file will be played in the browser.** The wording may vary.
- Set the **viewport dimensions to match your game's aspect ratio**, then **save the project page**.
- Open the project page and test the embedded game. Check the title screen and Instructions → Back flow if included, gameplay, movement, shooting, collision, scoring, and any timer or Game Over features you added.
- **Return to the [CISC 3667 F26 Unity Labs itch.io Game Jam page](#) and submit or add your project to the jam**, following its submission instructions.
- **Submit your playable itch.io project URL to Brightspace.**

Remember: upload the Web build ZIP, not the editable Unity project. Keep a backup of your Unity project so you can continue making changes.

Final project checklist

Area	Required completion
Lab 1	Background, Player, and Balloon are visible. Player movement and automatic Balloon movement work. The Pin Prefab is saved.
Lab 2: Shooting	Space creates a Pin Prefab instance. Pins move upward and are removed off-screen.
Lab 2: Collision	Pin and Balloon have the appropriate Collider2D/trigger setup, and at least one has a Rigidbody2D. Only Pin hits trigger the pop response.
Lab 2: Feedback and score	The Balloon visibly pops or briefly disappears, the Pin disappears, and the score increases once per hit. Sound is optional.
Lab 2: Repeatable play	The Balloon returns or another Balloon appears so the player can continue playing.
Optional features	If included the title screen, instructions, timer, and Game Over flow work correctly. These features do not replace the required gameplay mechanics.
Web build	The game is built for Web and runs in a browser. The ZIP contains the complete build with index.html at the top level.
Submission	Project submitted to the CISC 3667 F26 Unity Labs itch.io Game Jam, and the playable itch.io URL submitted to Brightspace.